iDSP 2.0 Programmer's Guide

Item ID	PT-001004-A-1
Item type	Document
Item title	iDSP 2.0 Programmer's Guide
Authors	Philip Thrift, James Overturf, Ajai Narayan, Schuyler Patton, Todd Killain
Revision	1
Date	04 October, 2000
Supercedes items	n/a
Key changes wrt prev release	n/a
Has embedded or requires items	n/a

This is a preliminary document. The final version of this document and API specifications will be issued with the iDSP SDK 2.1 software release. The final version will also contain QoS-related features.

APPENDIX

TABLE OF CONTENTS

1. Introduction

This document provides a guide to developing iDSP 2.0 applications and components.

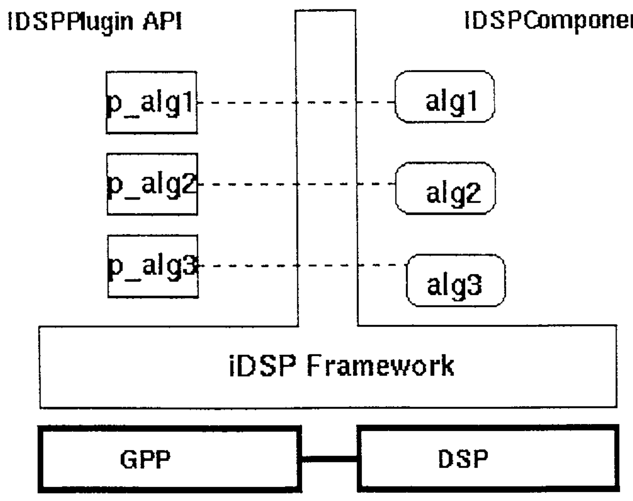
iDSP is a real-time platform for integrating GPP (General Purpose Processor) streaming media applications and/or players and DSP media algorithms (e.g. codecs, transforms, renderers, capturers, sources, sinks).

iDSP:

- provides a media-domain framework that is compliant with the eXpressDSP Algorithm Standard[1][2]
- processes and manages multiple, concurrent media streams
- provides real-time media processing with guaranteed in-box QOS

iDSP development is split into two parts:

- On the GPP-side, iDSP provides proxy objects (IDSPPlugins) corresponding to DSP algorithms (IDSPComponents). These proxy objects are accessed by the media player to accelerate media processeing of streams.
- On the DSP-side, iDSP provides an interface for developing and integrating DSP media algorithms (IDSPComponents).



Some typical media processing algorithms are:

- decoders and encoders
- color-space converts
- effect filters
- multiplexors and demultiplexors
- renderers and capturers

An iDSP *image* consists of a configuration of IDSPComponents (which can be split onto multiple DSPs). The iDSP runtime can schedule the execution of concurrent IDSPComponents in the image to meet time-constraints of the media player.

The iDSP SDK 2.1 release will contain an implementation of the IDSP 2.0 framework for Windows CE 2.12 with the Microsoft DXPAK media toolkit and a baseline set of IDSPComponents. Section 3 shows how the iDSP SDK release is used to create a DSP-accelerated media player with DXPAK. The iDSP architecture, however, is independent of any specific GPP media framework and GPP RTOS.

All interfaces in the iDSP API are specified using ANSI C, and are encapsulated in a collection of *types*. The following naming convention is used throughout.

- All type names have the form IDSP*TypeName*.

  Examples: IDSPPlugin, IDSPPluginManager

- All function names associated with a type have the form IDSP*TypeName_functionName*.

  Example: IDSPPluginManager_createPlugin

- All constant names associated with a type have the form IDSP*TypeName_CONSTANT*_NAME.

  Example: IDSPComponent_ALG_COMPLETED

- All global variable names associated with a type have the form IDSP*TypeName_GlobalName*.
- All field variables of a type are of the form *varName*.

The iDSP API Reference for all iDSP types can be found in section 9.

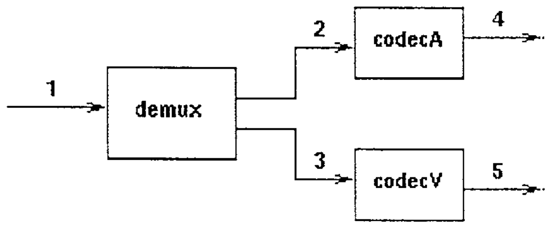
2. Application Developers: iDSP Plugin API

This section describes the iDSP Plugin API.

The iDSP Plugin API is the GPP-side interface to iDSP. Media application and player developers use this API to

- load a new DSP-side iDSP Framework image (a runtime collection of IDSPComponents)
- create IDSPPlugin objects (GPP-side proxies of IDSPComponents)
- manage dataflow on media tracks and synchronization of IDSPComponents

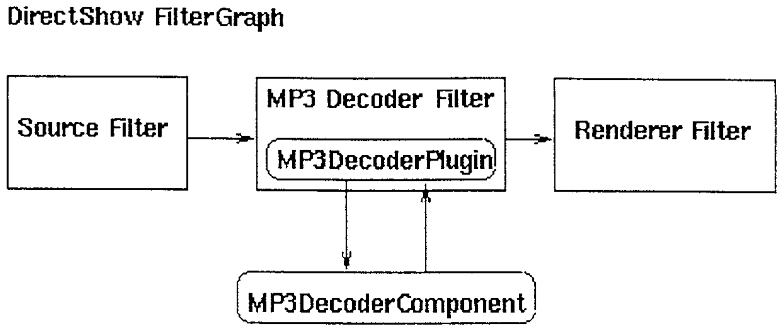
IDSPPlugin objects process media buffers on input and output *tracks*. IDSPPlugins can be connected into dataflow graphs as shown below:



The application developer has access to the dataflow on tracks with the *issue/reclaim* model described below. Media tracks are logical directed channels that carry buffers of media data.

The iDSP Plugin API allows application developers to build directed dataflow graphs of connected iDSP Components. The connections are constrained by the formats of the respective component's inputs and outputs.

Media application/developers use the iDSP Plugin API to create instances of DSP components to plug into their own application framework. The figure below shows a Microsoft DirectShow Filter Graph where the implementation of the audio decoder is based on an IDSPPlugin object created with the Framework API. The DSP MP3DecoderComponent provides the decoding function. In the figure below, MP3DecoderPlugin runs on the GPP and MP3DecoderComponent



runs on a DSP.

These are the iDSP API types used in the iDSP Plugin API:

- IDSPParams
- IDSPCommand
- IDSPFormat
- IDSPAudioFormat
- IDSPVideoFormat
- IDSPPlugin
- IDSPPluginManager

These types are documented in the iDSP API Reference (see section 9).

The interfaces are provided by including `<idsppim.h>` , `<idspplugin.h>, and` `<idspcomponent.h>` in an application.

Thes next section shows an example of using the iDSP Plugin API.

3. iDSP Plugin Example: Accelerating Windows Media

This section describes how to use the iDSP Plugin API to accelerate Windows Media.

Lines of code pertaining to iDSP Framework Plugin API are printed in bold.

The sample is based on DXPAK 1.0 and Windows CE 2.12. DXPAK is the DirectX Platform Adaptation Kit and includes DirectDraw, DirectSound, and DirectShow[1]. This document provides an overview on creating a DirectShow filter that uses a G.723 audio decoder component in iDSP. How-to information on using the Microsoft development tools for Windows CE is beyond the scope of this document. Please consult the Microsoft documentation as necessary.

3.1. Required Software

For the development workstation you will need Microsoft Windows NT 4.0 or later, Microsoft Windows CE Platform Builder 2.12, and the iDSP Framework runtime software. When installing Platform Builder you will need to install support for the target platform hardware – currently iDSP is only supported on the R4300 family of processors. Besides Platform Builder, you will also need to install DXPAK 1.0.

3.2. Required Hardware

For the target platform you will need a Windows CE platform onto which the iDSP Framework runtime has been ported – currently the ATI STWII reference platform. For the development platform, consult the Windows CE Platform Builder documentation for its hardware requirements.

3.3. Overview of the Sample

The sample consists of a DirectShow decoder filter that uses a G.723 audio decoder via the iDSP Plugin API and a sample application that plays the audio. Once the decoder filter DLL is installed in CE then Windows Media Player can be used as the player application if available on the target platform.

3.4. Steps for Creating the DirectShow Filter

The G.723 decoder is basically a DirectShow transform Filter so it derives from the DirectShow CTransformFilter class[2]:

[1] http://www.microsoft.com/Windows/embedded/ce/guide/features/dxpak_faq.asp
See also http://www.microsoft.com/DirectX/dxm/help/ds/c-frame.htm
[2] IDSP Framework code is shown in bold. All non-bold code is Microsoft DirectShow specific.

```
class CG723DecoderFilter : public CTransformFilter
```

The CtransformFilter class has a number of methods that must be overridden to implement the filter. You will have to override the following methods.

```
HRESULT CheckInputType(const CMediaType* mtIn);


HRESULT CheckTransform(const CMediaType *mtIn

                      , const CMediaType *mtOut);

HRESULT GetMediaType(int iPosition, CMediaType *pMediaType);

HRESULT DecideBufferSize(IMemAllocator *pAlloc

                        , ALLOCATOR_PROPERTIES *pProperties);

HRESULT Transform(IMediaSample *pIn, IMediaSample *pOut);
```

Consult the DirectShow documentation for detailed explanations of these functions.

Another function implemented in the decoder filter is

```
HRESULT Copy(IMediaSample *pSource, IMediaSample *pDest) const;
```

This function copies the DirectShow header information, e.g. time stamps, from the source to the destination media samples.

The CheckInputType, CheckTransform, DecideBufferSize, and GetMediaType functions enable DirectShow to determine if your decoder filter is appropriate to connect to an instantiated source filter. The Transform function is called when an IMediaSample is ready for decoding and it uses the Copy function to make sure DirectShow header information is relayed to the output IMediaSample.

The IMediaSample object is a DirectShow object that wraps a media buffer. Besides the raw bytes for the media sample, the IMediaSample interface keeps track of timestamps and other information needed by the DirectShow framework.

The following member variables are defined to interact with the iDSP Framework.

```
IDSPParams m_pluginParams;   // the Plugin's attributes

IDSPPlugin *m_pPlugin;       // the iDSP Plugin object
```

3.5. Filter setup and instantiation

To implement the decoder filter, the following structures must be defined to properly define the filter's pins.

```
const AMOVIESETUP_MEDIATYPE sudInPinType =
{
    &MEDIATYPE_Audio,       // Major type
    &MEDIASUBTYPE_NULL           // Minor type
};
```

sudInPinType defines the media type for the filter's input pin

```
const AMOVIESETUP_MEDIATYPE sudOutPinType =
{
    &MEDIATYPE_Audio,       // Major type
    &MEDIASUBTYPE_PCM       // Minor type
};
```

sudOutPinType defines the media type for the filter's output pin.

```
const AMOVIESETUP_PIN sudpPins[] =
{
    { L"Input",             // Pins string name
      FALSE,                // Is it rendered
      FALSE,                // Is it an output
      FALSE,                // Are we allowed none
      FALSE,                // And allowed many
      &CLSID_NULL,          // Connects to filter
      NULL,                 // Connects to pin
      1,                    // Number of types
      &sudInPinType         // Pin information
    },
     { L"Output",            // Pins string name
      FALSE,                // Is it rendered
      TRUE,                 // Is it an output
      FALSE,                // Are we allowed none
      FALSE,                // And allowed many
      &CLSID_NULL,          // Connects to filter
      NULL,                 // Connects to pin
      1,                    // Number of types
      &sudOutPinType        // Pin information
    }
};
```

sudpPins defines the DirectShow filter's pins. This structure is used with the following structure for automated registration of DirectShow filter objects and for instantiating the filter object at runtime.

```
const AMOVIESETUP_FILTER sudDSPFilter =
{
    &CLSID_G723DecoderFilter,         // Filter CLSID
    L"iDSP G.723 Decoder",               // String name
    MERIT_PREFERRED,                // Filter merit
    2,                               // Number of pins
    sudpPins                         // Pin information
};
```

sudDSPFilter defines the GUID, name, and pins for the DirectShow filter. The GUID was created using the Microsoft GUIDGEN tool and is defined as:

```
// {4CBF66C0-D7DE-11d3-BBAC-0050049FCFDA}
DEFINE_GUID(CLSID_G723DecoderFilter,
        0x4cbf66c0, 0xd7de, 0x11d3, 0xbb, 0xac, 0x0, 0x50, 0x4,
        0x9f, 0xcf, 0xda);
```

```
GUIDDEF CLSID_G723DecoderFilter;
```

The MERIT_PREFERRED parameter in the sudDSPFilter above sets the Filter's merit value to the highest level. This merit value is used by DirectShow to determine what filter gets instantiated when multiple filters are registered with the same input pin media types. By setting the merit to MERIT_PREFERRED the iDSP implemented DirectShow filter will have priority over all others (e.g. over the Microsoft filters that encode/decode on the GPP).

```
CFactoryTemplate g_Templates[] = {
    { L"iDSP G.723 Decoder",
      &CLSID_G723DecoderFilter,
      CG723DecoderFilter::CreateInstance,
       NULL,
      &sudDSPFilter }
};
```

The g_Templates structure provides a link between the OLE entry point in the filter's DLL and the factory method used to instantiate the filter object. For the sample, the factory method is a static method called CreateInstance().

```
    int g_cTemplates = sizeof(g_Templates) / sizeof(g_Templates[0]);
```

The g_cTemplates variable is used by DirectShow during the instantiation process.

To complete the self registration for a DirectShow filter you need to implement the following functions:

```
STDAPI DllRegisterServer(void)
{
    // registers object, typelib and all interfaces in typelib
    return AMovieDllRegisterServer2(TRUE);
}

///////////////////////////////////////////////////////////////////
// DllUnregisterServer - Removes entries from the system registry

STDAPI DllUnregisterServer(void)
{
    return AMovieDllRegisterServer2(FALSE);
}
```

Now the filter object can be registered using the RegSvr32.exe program (in Windows), or RegSvrCE (in Windows CE MSTV). In a standard CE 2.12 build with DXPAK, no registration program is available so you must manually change the Windows CE registry (see below).

When the filter is instantiated COM uses the factory method CG723DecoderFilter::CreateInstance() as initialized in the g_Templates structure above. This factory method and the filter's constructor and destructor are implemented as follows.

```
CUnknown *CG723DecoderFilter::CreateInstance(LPUNKNOWN punk
                                            , HRESULT * phr)
  {
    CG723DecoderFilter *pNewObject;

    pNewObject = new CG723DecoderFilter(NAME("iDSP G.723 Decoder")
                                       , punk
                                       , phr);
    if (pNewObject == NULL) {
        *phr = E_OUTOFMEMORY;
    }
    return pNewObject;
} // CreateInstance()


CG723DecoderFilter::CG723DecoderFilter(TCHAR *tszName
                                       , LPUNKNOWN punk
                                       , HRESULT *phr) :
    CTransformFilter(tszName, punk, CLSID_G723DecoderFilter)
{
    int result;
    *phr = NOERROR;
    m_pluginParams.input_formats = NULL;
    m_pluginParams.output_formats = NULL;
    m_pPlugin = NULL;

    // connect to the iDSP Framework
    result = IDSPFramework_initialize();
    if (IDSP_Failed(result)) {
        *phr = E_FAILED;
        return;
    }
}

CG723DecoderFilter::~CG723DecoderFilter()
{
    if (m_pPlugin) {
        IDSPPluginManager_deletePlugin(m_pPlugin);
        if (m_plugin.m_pluginParams.input_formats) {
            delete m_plugin.m_pluginParams.input_formats;
        }
        if (m_plugin.m_pluginParams.output_formats) {
            delete m_plugin.m_pluginParams.output_formats;
        }
        delete m_pPlugin;
    }
}
```

3.6. CtransformFilter implementation

The following functions are implementations of abstract functions in the CtransformFilter class.

```
HRESULT CG723DecoderFilter::CheckInputType(const CMediaType* mtIn)
{
    // check this is a WaveFormatEx type
    if (*mtIn->FormatType() != FORMAT_WaveFormatEx) {
        return E_INVALIDARG;
    }

    // Check the media type
    if (IsEqualGUID(*mtIn->Type(), MEDIATYPE_Audio) )
    {
        // Check for 8KHz, 0 bit
        WAVEFORMATEX *wfm = (WAVEFORMATEX*)(mtIn->Format());
        if ( (wfm->nSamplesPerSec == 8000)
             && (wfm->wBitsPerSample == 0)
             && (wfm->wFormatTag == 273) )
        {
            count++;
            return NOERROR;
        }
    }

    return E_FAIL;
}
```

The CheckInputType() method is used by DirectShow when connecting filters in the filter graph. In order for the output pin of one filter to connect to the input pin of another, the two pins must have an exact match for the media sample. Often, a filter will only register a major type and sub type. For the G.723 decoder the major type is MEDIATYPE_Audio and the subtype is FORMAT_WaveFormatEx. However, a WAVEFORMATEX structure can denote wildly different content (e.g. 44.1KHz, 16Bit, stereo PCM data).

```
HRESULT CG723DecoderFilter::CheckTransform(const CMediaType *mtIn
                                          , const CMediaType *mtOut)
{
    HRESULT hr;
    if (FAILED(hr = CheckInputType(mtIn)))
        return hr;

    // format must be a WaveFormatEx
    if (*mtOut->FormatType() != FORMAT_WaveFormatEx)
          return E_INVALIDARG;

    // formats must be big enough
    if (mtIn->FormatLength() < sizeof(WAVEFORMATEX) ||
          mtOut->FormatLength() < sizeof(WAVEFORMATEX))
          return E_INVALIDARG;

    return NOERROR;
}
```

For the G.723 encoded samples, the nSamplesPerSec filed is set to 8000 (8KHz) , the wBitsPerSample is set to 0 (0Bit indicating a compressed format), and the wFormatTag is set to 273.

The CheckInputType() method returns NOERROR (indicating success) if a G.723 media format is passed in and E_FAIL (indicating failure) otherwise. This guarantees that only a filter that claims to output G.723 encoded data will be able to connect to our decoder filter. In general a transform filter may support multiple media sample types and CheckInputType must return NOERROR if any supported type is passed in.

A transform filter may support multiple output pin media types as well and the CheckTransform() function checks that the filter can handle the transformation between a given pair of supported types. This check enables a filter to support multiple input and output types without necessarily supporting every possible combination of input/output type pairs.

```
HRESULT CG723DecoderFilter::GetMediaType(int iPosition
                                        , CMediaType * pMediaType)
{
    // Is the input pin connected?
    if (m_pInput->IsConnected() == FALSE)
       return E_UNEXPECTED;

    // This should never happen
    if (iPosition < 0)
        return E_INVALIDARG;

    // we have only one item to offer
    if (iPosition > 0)
        return VFW_S_NO_MORE_ITEMS;

    //pMediaType = new CMediaType(&MEDIATYPE_Audio);
    pMediaType->SetType(&MEDIATYPE_Audio);
    pMediaType->SetSubtype(&MEDIASUBTYPE_PCM);
    pMediaType->SetFormatType(&FORMAT_WaveFormatEx);
    WAVEFORMATEX *wfm = (WAVEFORMATEX*)(pMediaType->Format());
    if (wfm == NULL)
       wfm = new WAVEFORMATEX;
    wfm->cbSize = 0;
    wfm->nAvgBytesPerSec = 16000;
    wfm->wBitsPerSample = 16;
    wfm->nSamplesPerSec = 8000;
    wfm->wFormatTag = WAVE_FORMAT_PCM; // wave format
    wfm->nChannels = 1;
    wfm->nBlockAlign = 2;
    pMediaType->SetFormat((BYTE*)wfm, sizeof(WAVEFORMATEX));
    return NOERROR;
}
```

The GetMediaType() function enables DirectShow to automatically connect filters together by allowing the framework to query your filter for all of its output pin media types. For example, after your decoder filter has been connected to the graph, the FilterGraphManager can call GetMediaType() to get one output media type. It will then look in the registry for a filter with a matching input pin media type. In our case it will find the standard Microsoft Wavedev Audio Renderer which can accept major type MEDIATYPE_Audio and subtype MEDIASUBTYPE_PCM. The FilterGraphManager will instantiate the renderer and pass it the CMediaType object which contains a FORMAT_WaveFormatEx type with an audio format of 8KHz, 16Bit, mono, PCM. The FilterGraphManager calls the CheckInputType() function on the audio renderer which returns success so the FilterGraphManager connects the pins together.

```
HRESULT CG723DecoderFilter::DecideBufferSize(IMemAllocator *pAlloc
                                 , ALLOCATOR_PROPERTIES * pProperties)
{
    // Is the input pin connected?
    if (m_pInput->IsConnected() == FALSE) {
        return E_UNEXPECTED;
    }

    ASSERT(pAlloc);
    ASSERT(pProperties);
    HRESULT hr = NOERROR;

    pProperties->cBuffers = 1;
    pProperties->cbBuffer = 480;
    ASSERT(pProperties->cbBuffer);

    // Ask the allocator to reserve us some sample memory, NOTE
    // the function can succeed (that is return NOERROR) but still
    // not have allocated the memory that we requested, so we must
    // check we got whatever we wanted
    ALLOCATOR_PROPERTIES Actual;
    hr = pAlloc->SetProperties(pProperties,&Actual);
    if (FAILED(hr)) {
        return hr;
    }

    ASSERT( Actual.cBuffers == 1 );

    if (pProperties->cBuffers > Actual.cBuffers ||
        pProperties->cbBuffer > Actual.cbBuffer) {
        return E_FAIL;
    }

    IDSPAudioFormat *inpformat, *outformat;
    inpformat = new IDSPAudioFormat;
    inpFormat->format.data_type = IDSPAudioFormat_PARSED;
    inpformat->format.format_name = IDSPSTRING("G.723");
    inpformat->audio_type = IDSPAudioFormat_COMPRESSED;
    inpformat->microseconds_per_frame = 30000;
    m_pluginParams.input_formats = inpformat;

    outformat = new IDSPAudioFormat;
    outformat->format.data_type = IDSPAudioFormat_PARSED;
    outformat->format.format_name = IDSPSTRING("PCM");
    outformat->audio_type = IDSPAudioFormat_PCM;
    outformat->microseconds_per_frame = 30000;
    outformat->sample_rate = 8000;
    outformat->number_channels = 1;
    outformat->bits_per_sample = 16;
    m_pluginParams.output_formats = outformat;

    m_pluginParams.format_type = IDSPPlugin_AUDIO;
    m_pluginParams.plugin_type = IDSPPlugin_CODEC;
    m_pPlugin = IDSPPluginManager_createPlugin(&m_pluginParams);
    if (!m_pPlugin)
        return E_FAIL;

    return NOERROR;
}
```

DecideBufferSize() is used by DirectShow to determine how large a buffer to allocate for the decoder filter's output. The IMemAllocator is created by the downstream filter (the renderer in our case) and is responsible for actually allocating the buffer space. For G.723, the output buffer size is always 480 bytes which represents a 30ms sample on a 6Kbps input stream.

```
HRESULT CG723DecoderFilter::Transform(IMediaSample *pIn
                                     , IMediaSample * pOut)
{
    long size = 0;

    // Copy the properties across
    HRESULT hr = Copy(pIn, pOut);
    if (FAILED(hr))
        return hr;

    // Get the buffer pointers
    BYTE *pInData, *pOutData;
    pIn->GetPointer(&pInData);
    pOut->GetPointer(&pOutData);

    // Initialize the input parameters
    m_pOrb->m_inputArg[0]->size = pIn->GetActualDataLength();
    m_pOrb->m_inputArg[0]->data = pInData;

    if (m_pOrb->m_inputArg[0]->size == 0)
          return S_OK;

    m_pOrb->m_outputArg[0]->size = pOut->GetSize();
    m_pOrb->m_outputArg[0]->data = pOutData;

    // issue the buffers for input/output
    IDSPPlugin_issue(m_pPlugin, IDSPPlugin_INPUT, 0, pInData
                     , pIn->GetActualDataLength());
    IDSPPlugin_issue(m_pPlugin, IDSPPlugin_OUTPUT, 0, pOutData
                     , pOut->GetSize());

    // reclaim the buffers
    // block until input buffer consumed
    IDSPPlugin_reclaim(m_pPlugin, IDSPPlugin_INPUT, 0, &pInData
                     , &size);
    // block until output buffer produced
    IDSPPlugin_reclaim(m_pPlugin, IDSPPlugin_OUTPUT, 0, &pOutData
                     , &size);

    pOut->SetActualDataLength(size);

    return NOERROR;
}
```

The Transform() is called by DirectShow when an upstream filter (in our case a source filter serving G.723 encoded data) sends a buffer to our decoder filter. The DirectShow framework allows you to assume as much control over the data flow as you want and it also provides basic data flow control for you. By only overriding Transform() we receive an input and output IMediaSample object and can allow DirectShow to manage receiving the input media sample and sending the output media sample.

Transform() uses a helper function Copy() to initialize the DirectShow header information on the IMediaSample object (e.g. its timestamp data).

```
HRESULT CG723DecoderFilter::Copy(IMediaSample *pSource
                                        , IMediaSample * pDest) const
{
     // Copy the sample data
    BYTE *pSourceBuffer, *pDestBuffer;
    long lSourceSize = pSource->GetActualDataLength();
    long lDestSize   = pDest->GetSize();

    ASSERT(lDestSize >= lSourceSize);

    pSource->GetPointer(&pSourceBuffer);
    pDest->GetPointer(&pDestBuffer);

     // Copy the sample times
    REFERENCE_TIME TimeStart, TimeEnd;
    if (NOERROR == pSource->GetTime(&TimeStart, &TimeEnd)) {
        pDest->SetTime(&TimeStart, &TimeEnd);
    }


     LONGLONG MediaStart, MediaEnd;
    if (pSource->GetMediaTime(&MediaStart,&MediaEnd) == NOERROR) {
        pDest->SetMediaTime(&MediaStart,&MediaEnd);
    }

    // Set the Sync point property
    pDest->SetSyncPoint(TRUE);


    // Copy the media type
    AM_MEDIA_TYPE *pMediaType;
    pSource->GetMediaType(&pMediaType);
     pDest->SetMediaType(pMediaType);
    DeleteMediaType(pMediaType);


    // Copy the preroll property
    pDest->SetPreroll(FALSE);

    // Copy the discontinuity property
    pDest->SetDiscontinuity(FALSE);

    // Copy the actual data length
    long lDataLength = pSource->GetActualDataLength();
     pDest->SetActualDataLength(lDataLength);

    return NOERROR;

} // Copy()
```

The implementation for the Copy() function is shown below. The *media* and *sample* start/end times are assigned by the source filter and merely copied to the output buffer. Other values, like the PreRoll property is set to a default value. The SyncPoint property is mainly used in video codecs to indicate an I-frame. For audio, every frame is technically a synchronization point so that property is set to true.

3.7. Steps for Creating a Player Application

The player application must load the iDSP Framework and instantiate the filter graph. We will assume that the input file is a windows media file containing G.723 encoded data called *g723Sample.asf.*

Globals and function prototypes for the application main function:

```
#define MAX_LOADSTRING 100

// Global Variables:
HINSTANCE hInst;                          // current instance
TCHAR szTitle[MAX_LOADSTRING];            // The title bar text
TCHAR szWindowClass[MAX_LOADSTRING];      // The title bar text

// Foward declarations of functions included in this code module:
//ATOM                         MyRegisterClass(HINSTANCE hInstance);
BOOL                           InitInstance(HINSTANCE, int);
LRESULT CALLBACK        WndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK        About(HWND, UINT, WPARAM, LPARAM);
```

The main function must initialize COM so that DirectShow objects can be instantiated and it must also initialize the iDSP Framework. The iDSP Framework is initialized by calling IDSPPluginManager_loadFramework(), which loads the DSP with the iDSP Framework, and then calling IDSPPluginManager_startFramework(), which initializes the frameowkr and starts its scheduling threads. For now iDSP is only implemented on a single DSP system so the processor id whould be defaulted to 0 in all iDSP functions.

The actual creation of the media player is handled by a helper class called CPlayer (described below).

The main function provides a simple message loop for handling windows messages. It uses the InitInstance(), WndProc, and About() functions for initializing the window and handling windows messages.

3.8. Implementation of the WinMain function

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance
                   , LPTSTR szCmdLine, int nCmdShow)
{
    MSG msg;
    HACCEL hAccelTable;
    HRESULT hr;
    int result;

    // Initialize global strings
    LoadString(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadString(hInstance, IDC_CEBITAPP, szWindowClass, MAX_LOADSTRING);

     // Perform application initialization:
    if (!InitInstance (hInstance, nCmdShow))
       return FALSE;

    hAccelTable = LoadAccelerators(hInstance, (LPCTSTR)IDC_CEBITAPP);


    // Initialize COM
    hr = CoInitializeEx( NULL, COINIT_MULTITHREADED );
    if(FAILED(hr)) {
          CoUninitialize();
          return FALSE;
    }

    result = IDSPPluginManager_initialize();


    if (IDSP_Failed(result)) {
       CoUninitialize();
       return FALSE;
     }


    result = IDSPPluginManager_loadFramework(
                                           IDSPSTRING("IDSPFramework.c6x"
                                           , 0);
    if (IDSP_Failed(result)) {
       CoUninitialize();
       return FALSE;
    }

    result = IDSPPluginManager_startFramework(0);
    if (IDSP_Failed(result)) {
       CoUninitialize();
       return FALSE;
    }

     CPlayer *player = new CPlayer();

    // Main message loop:
    while (GetMessage(&msg, NULL, 0, 0)) {
       if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg)) {
                  TranslateMessage(&msg);
                  DispatchMessage(&msg);
             }
      }

    delete player;
    result = IDSPPluginManager_stopFramework(0);
    CoUninitialize();  // Release COM

     return TRUE;
}
```

The InitInstance function defines the look and feel of the window. It initializes the windows properties like background color, size, and position.

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   HWND hWnd;
   WNDCLASS wndclass ;                                      // Window Class
Structure

   hInst = hInstance; // Store instance handle in our global variable

#ifndef _WIN32_WCE
      wndclass.style          = CS_HREDRAW | CS_VREDRAW;
      wndclass.lpfnWndProc    = WndProc;

      wndclass.cbClsExtra     = 0;

      wndclass.cbWndExtra     = 0;
      wndclass.hInstance      = hInstance;
      wndclass.hIcon          = 0;
      wndclass.hCursor        = 0;
      wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH);
      wndclass.lpszMenuName  = NULL;
      wndclass.lpszClassName = szTitle;
#else
      // Configure the Window Class
      wndclass.style          = 0;
      wndclass.lpfnWndProc    = WndProc;
      wndclass.cbClsExtra     = 0;
      wndclass.cbWndExtra     = 0;
      wndclass.hInstance      = hInstance;
      wndclass.hIcon          = 0;
      wndclass.hCursor        = 0;
      wndclass.hbrBackground = (HBRUSH) GetStockObject (BLACK_BRUSH);
      wndclass.lpszMenuName  = NULL;
      wndclass.lpszClassName = szTitle;
#endif

      // Register the Window Class
      if (!RegisterClass (&wndclass))
      {
            MessageBox (NULL, TEXT("WinMain(): RegisterClass() failed"),
                  TEXT("Err! - VIDEOAPP"), MB_OK | MB_ICONEXCLAMATION);
            return(FALSE);
      }

#ifndef _WIN32_WCE
      // Create the Window
      if (!(hWnd = CreateWindow ((LPCTSTR)szTitle,
(LPCTSTR)szWindowClass,
                              WS_OVERLAPPEDWINDOW,
                              CW_USEDEFAULT, CW_USEDEFAULT,
                              CW_USEDEFAULT, CW_USEDEFAULT,
                                          NULL, NULL, hInstance, NULL)))
      {
            return 0;
      }
#else
      // Initialize Common Controls and create the window
//    InitCommonControls();

      if (!(hWnd = CreateWindow ((LPCTSTR)szTitle, (LPCTSTR)szWindowClass,
                              WS_VISIBLE,
                              CW_USEDEFAULT, CW_USEDEFAULT,
                              CW_USEDEFAULT, CW_USEDEFAULT,
                                          NULL, NULL, hInstance, NULL)))
      {

            return 0;

      }

#endif



   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}
```

The WndProc() function is the main window message processing function.

```
//  WM_COMMAND     - process the application menu
//  WM_PAINT       - Paint the main window
//  WM_DESTROY     - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM
lParam)
{
      int wmId, wmEvent;
      PAINTSTRUCT ps;
      HDC hdc;
      TCHAR szHello[MAX_LOADSTRING];
      LoadString(hInst, IDS_HELLO, szHello, MAX_LOADSTRING);

      switch (message)
      {
            case WM_COMMAND:
                  wmId    = LOWORD(wParam);
                  wmEvent = HIWORD(wParam);
                  // Parse the menu selections:
                  switch (wmId)
                  {
                        case IDM_ABOUT:
                           DialogBox(hInst, (LPCTSTR)IDD_ABOUTBOX
                                    , hWnd, (DLGPROC)About);
                           break;
                        case IDM_EXIT:
                           DestroyWindow(hWnd);
                           break;
                        default:
                           return DefWindowProc(hWnd, message
                                                             , wParam, lParam);
                    }
                    break;
            case WM_PAINT:
                    hdc = BeginPaint(hWnd, &ps);
                    // TODO: Add any drawing code here...
                    RECT rt;
                    GetClientRect(hWnd, &rt);
                    DrawText(hdc, szHello, 42, &rt, DT_CENTER);

                    EndPaint(hWnd, &ps);

                    break;
            case WM_DESTROY:
                    PostQuitMessage(0);
                    break;
            default:
                    return DefWindowProc(hWnd, message, wParam, lParam);
   }
   return 0;
}
```

The About() function is a callback for handling messages on the "about" dialog box.

```
LRESULT CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    switch (message)
    {
            case WM_INITDIALOG:
                    return TRUE;

            case WM_COMMAND:
                    if (LOWORD(wParam) == IDOK || LOWORD(wParam) ==
IDCANCEL)
                    {
                            EndDialog(hDlg, LOWORD(wParam));
                            return TRUE;
                    }
                    break;

    }

    return FALSE;

}
```

3.9. Implementation of the CPlayer class

```
/* Cplayer.h */

class CPlayer
{
public:
      CPlayer();
      virtual ~CPlayer();

protected:
    IGraphBuilder *m_pGraph;                 // The Filter Graph Manager

private:

      BOOL InitializeGraph();

      void PlayGraph();
};
```

The constructor creates an instance of the DirectShow FilterGraphManager object to retrieve its IGraphBuilder interface. The constructor then creates the graph to play the G.723 file (InitilizeGraph()) and then starts the graph playing.

```
/* CPlayer.cpp */

CPlayer::CPlayer() :
            m_piV1Filter(NULL)
      ,     m_piV2Filter(NULL)
{
      HRESULT hr = CoCreateInstance( CLSID_FilterGraph
                                    , NULL,
                                    , CLSCTX_INPROC
                                    , IID_IGraphBuilder
                                    , (void**)&m_pGraph);
    if (FAILED(hr)) {
            m_pGraph = NULL;
            return;
    }

     if (!InitializeGraph())
        return;


    PlayGraph();
}

CPlayer::~CPlayer()
{
     if (m_pGraph)
        m_pGraph->Release();
}
```

```
BOOL CPlayer::InitializeGraph()
{
    HRESULT hr;

    hr = m_pGraph->RenderFile(TEXT("G723Sample.asf"), NULL);
     if (FAILED(hr))
        return FALSE;

    return TRUE;
}
```

InitializeGraph() creates the filter graph needed to play the G.723 file and currently just creates the G.723 graph. PlayGraph() simply instructs the FilterGraphManager to start playing.

```
void CPlayer::PlayGraph()
{
    HRESULT   hr;
    IMediaControl *pMC;

    // Obtain the interface to our filter graph.
    hr = m_pGraph->QueryInterface(IID_IMediaControl, (void **) &pMC);

     if( SUCCEEDED(hr) ){
        // Ask the filter graph to play and release the interface.
        hr = pMC->Run();
        pMC->Release();
    }
}
```

4. Algorithm Developers: iDSP Component API

This section describes the iDSP Component API and how to develop IDSPComponents for iDSP.

Many of the iDSP Component APIs are derived directly by extending the structures and objects defined in TI's expressDSP Algorithm Standard, referred to as the **XDAIS** standards [1,2]. Hence, it is necessary for the component programmer to have a good knowledge of that standard before using these APIs.

The iDSP Component API consists of the following types:

- IDSPResult
- IDSPComponent
- IDSPComponentFxns
- IDSPCommand
- IDSPParams

These types are specified in the iDSP API Reference in section 8.

Developing an IDSPComponent involves:

1. requesting persistent and scratch memories for algorithm execution.
2. implementing a set of interface functions called IALG_Fxns for algorithm control and execution.

The TI expressDSP standard (XDAIS) specifies the mechanism and the structure by which algorithms can request scratch and persistant memories from the framework. The structure is called memTab, a shortform for memory table. The definition of memTab structure is shown below. Please refer to the XDAIS standards documents for the description of the fields in this structure.

The 'base' for the allocated buffer is filled by the framework after it has allocated the memory space.

```
typedef struct IALG_MemRec {
   int             size;      /* size in MAU of allocation   */
   int             alignment;      /* alignment requirement (MAU) */
   IALG_MemSpace   space;      /* allocation space            */
   IALG_MemParams  params;     /* memory attributes           */
   void             *base;     /* base-addr. of allocated buf */
} IALG_MemRec;
typedef  IALG_MemRec   memTab[];
```

The IDSPComponent mandates that the component programmer requests **internal (on-chip)** memory using the memTab mechanism for the following structures:

1. The *algorithm stack*. This should be the VERY LAST entry in the table of memory requests (i.e, memTab[0]).
2. *Contextual data* required for correct operation of the algorithm.
3. All *look-up tables* as a single memory chunk.

The entries that go into requesting memTabs for the aforementioned structures are shown below. It is recommended that these requests are instantiated in the (algAlloc*) function.

```
memTab[0].size      = sizeof(CODEC_TI_Obj);
memTab[0].alignment = 0x10;
memTab[0].space     = IALG_INTERNAL;
memTab[0].params    = IALG_PERSIST;

memTab[1].size      = sizeof(CODEC_TI_Tables);
                      /* (448*2) + 384 + 384 + 556 */
memTab[1].alignment = 0x10;
memTab[1].space     = IALG_INTERNAL;
memTab[1].params    = IALG_SCRATCH;

memTab[2].size      = sizeof(stack);
memTab[2].alignment = 0x10;
memTab[2].space     = IALG_INTERNAL;
memTab[2].params    = IALG_PERSIST;
```

The relationship between the structure IALG_Fxns and its implementation is summarized below. The XDAIS standard defines this structure with a standard set of function pointers. It is the responsibility of the IDSPComponent Programmer to implement each of these functions. If the component does not require some of these functions, a function stub with NO executable code can be implemented.

```
/*=== IALG_Fxns (Defined in ialg.h, provided by XDAIS standard)
* This structure defines the fields and methods that must be
* supplied by all XDAS algorithms.
* implementationId  - unique pointer that identifies the module
                      implementing this interface.
* algActivate()     - NULL for IDSPComponent
* algAlloc()       - apps call this to query the algorithm about
*                     its memory requirements. Must be non-NULL.
* algControl()      - NULL for IDSPComponent
* algDeactivate()   - NULL for IDSPComponent
* algFree()         - query algorithm for memory to free when
*                     removing an instance. Must be non-NULL.
* algInit()         - The framework calls this to allow the
*                     algorithm to initialize memory requested
*                     via algAlloc(). Must be non-NULL.
* algMoved()          - NULL for IDSPComponent
* algNumAlloc()       - query algorithm for number of memory
*                       requests. May be NULL; NULL => number of
*                       memrecs less then IALG_DEFMEMRECS.
*============================================================*/
typedef struct IALG_Fxns {
    Void    *implementationId;
    Void    (*algActivate)(IALG_Handle);
    Int     (*algAlloc)(const IALG_Params *, struct IALG_Fxns **,
                                                  IALG_MemRec *);
     Int     (*algControl)(IALG_Handle, IALG_Cmd, IALG_Status *);
    Void    (*algDeactivate)(IALG_Handle);
    Int     (*algFree)(IALG_Handle, IALG_MemRec *);
    Int     (*algInit)(IALG_Handle, const IALG_MemRec *,
                                 IALG_Handle, const IALG_Params *);
    Void    (*algMoved)(IALG_Handle, const IALG_MemRec *,
                                 IALG_Handle, const IALG_Params *);
    Int     (*algNumAlloc)(Void);
} IALG_Fxns;
```

One possible way of implementing the aforementioned functions and binding them to the function pointers is shown below:

```
/*module_vendor_interface */
IDSPComponent_Fxns CODEC_TI_ICODEC = {
            &CODEC_TI_IALG,          /* module ID       */
            NULL,                    /* algactivate     */
            CODEC_TI_alloc,          /* algAlloc */
            NULL,       /* control (NULL=>no control ops) */
            NULL                             /* algdeactivate   */
            CODEC_TI_free,                   /* algfree         */
            CODEC_TI_initObj,        /* alginit         */
            NULL,          /* moved (NULL=>not supported)*/
            CODEC_TI_numAlloc/*numAlloc (NULL=>IALG_DEFMEMRECS) */

            /* Extended specifically for IDSPComponent */
            CODEC_TI_processbuffers,
            CODEC_TI_componentControl
 };
```

A separate 'C' file (ex. CODEC_TI.c) may contain the implementations of all the bound functions as follows:

```
Void CODEC_TI_initObj(IALG_Handle handle)
{
     /*   IDSPComponent Programmer implements algActivate !   */
}

Int CODEC_TI_alloc( const IALG_Params  *codecParams,   IALG_Fxns  **fxns,
                                                      IALG_MemRec  memTab[] )
{
   /* IDSPComponent Programmer implements algAlloc!   */
}

Int CODEC_TI_free(IDSPComponent* alg, IALG_MemRec* memTab);
```

```
{
    /* IDSPComponent Programmer implements algFree!    */
}

Int CODEC_TI_numAlloc ( );
{
    /* IDSPComponent Programmer implements algNumAlloc!    */
}

Int CODEC_TI_processbuffers ( IDSPComponent*  alg,   IDSPRingBuffer*   inbuf,
                              IDSPRingBuffer*   outbufs,  IDSPResult* result )
{
    /*   IDSPComponent Programmer implements processbuffers !   */
}

Int CODEC_TI_componentControl ( IDSPComponent* alg,  IDSPCommand* cmd,
                                                      IALG_Status* status)
{
    /*   IDSPComponent Programmer implements componentControl !   */
}

...
```

This mechanism binds the implementation to the function pointers. The framework itself accesses the implementations through the function pointers.

5. IDSP Component Test Environment

The environment for testing an iDSP software component to ensure compliance with the mandatory requirements of the iDSP-Platform (enumerated in Section IV) is the following:

Hardware Requirements:

- Pentium based PC or equivalent running Windows-2000 or NT.
- C6202 PCI card from Blue Wave Systems, BWS Part No: PCI/C6202-EVM.
- JTAG controller from TI

  http://www.micro.ti.com/asp/dsp/operations/planning/tools

  Part Nos: TMDS3080002 & TMDS30510

Software Requirements:

- Code-Composer Studio 1.2 (CCS-1.2) code development package from TI.

  http://dspvillage.ti.com/docs/tools/dsp/ccs/index.htm

- Microsoft Visual C++ V 6.0.

The component runs in the scenario shown in Fig. 1. The test-input data is in a file on the PC hard-disk and the output data also goes into a file on the PC hard-disk. The executable code (COFF file) is loaded also from a file on the PC into the C6x external memory using CCS-1.2 and the JTAG interface. The overall testing mechanism is described below.

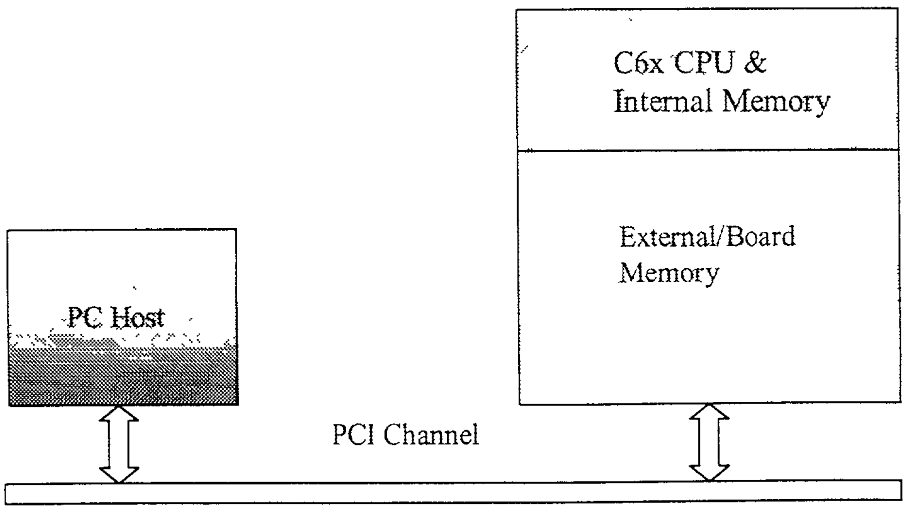


Fig. 1

The test environment defines a 'main' (in C) or equivalent. The 'main' simulates the iDSP-Platform framework, and does the following for the component under test:

1. It declares two ring-buffers *(input & output)* in the external C6 memory with **global access.** The interface structure for the ring-buffers is defined in the iDSP API Reference.
2. It creates an instance of the algorithm using **TI XDAIS standard functions (ALG_create).**
3. It initializes the algorithm again using **TI XDAIS standard functions (ALG_init, ALG_alloc).**
4. It then starts a 'while-loop' for simulating a continuous operation of the algorithm.
5. It fills input ring-buffer (not necessarily fully) with the data from the input file (on the PC) using fopen, fread etc. This simulates the data input, which is performed by the iDSP platform in the production system.
6. It invokes the DSP algorithm (ex: encode, decode, filter etc) using the function *processbuffers( )* defined in the structure IDSPComponentFxns, which extends the IALG_Fxns (a set of base functions defined in **TI XDAIS standard functions).** The algorithm processes the data in the input ring-buffer and fills the output ring-buffer with its output. This 'while-loop'- operation of the component (for example, codec) continues until an exception or orderly exit is reached.
7. It empties the data from the output ring-buffer (not necessarily completely) to an output file (on the PC) using fopen, fwrite etc. This simulates the data output, which is performed by the iDSP platform in the production system.

Taking all of this into account, a sample pseudo-code for the 'main' function of your test environment is shown below:

```
#include <stdio.h>
#include "idspcomponent.h"
#include "idsprbuf.h"
#include "ialg.h"
#include "alg.h"

unsigned char input_buffer[INPUT_RING_BUFFER_SIZE];
unsigned char output_buffer[OUTPUT_RING_BUFFER_SIZE];

int main()
{
  ALG_Handle          handle;
  CODEC_TI_Handle     obj;
  int                       i;
  FILE                      *infile, *outfile;
  IDSPRingBuffer            *inbuf, inbuf0, *outbuf, outbuf0;
  IDSPResult                *result, result0;

  char input_file [100]   = "INPUT_FILENAME" ;
   char output_file [100] = "OUTPUT_FILENAME" ;

 inbuf  = &inbuf0;       result = &result0;
/*create an instance of the algorithm object ... use default parameters */
handle = ALG_create( (IALG_Fxns *)&IMPLEMENTATION_ID,
                    (IALG_Handle)NULL, (IALG_Params *)NULL);
if (handle == NULL) {
      printf ("object creation failed\n");
      exit(-1);
}
obj = (CODEC_TI_Handle)handle;

/*  Opening files for Reading & Writing
*/
infile    = fopen( input_file, "rb");
outfile   = fopen( output_file "wb");

/*  Ring-Buffer Initializations
*/
inbuf->LOW_MEM_PTR  = input_buffer;
inbuf->HIGH_MEM_PTR = input_buffer + INPUT_RING_BUFFER_SIZE;
inbuf->DATA_START      = input_buffer;
inbuf->DATA_END          = input_buffer;

outbuf->LOW_MEM_PTR  = output_buffer;
outbuf->HIGH_MEM_PTR = output_buffer + OUTPUT_RING_BUFFER_SIZE;
outbuf->DATA_START      = output_buffer;
outbuf->DATA_END          = output_buffer;

/* Processing Loop
 */                                                    */
while (1)
{
      Required manipulations to keep input & output ring-buffers operating
correctly

      if( !feof(bsinfile) )    fread( inbuf->DATA_END, RING_BUFFER_SIZE, 1,
bsinfile );

/* The processbuffers Call
*/
     IMPLEMENTATION_ID.processbuffers ((IDSPComponent *)handle,
                                        (IDSPRingBuffer*) inbuf->DATA_START,
                                        (IDSPRingBuffer*)outbuf->DATA_START,
                                        (IDSPResult*)result );

     fwrite(outbuf->DATA_START, size, result->ouputProduced, out_file);

/* Required manipulations to keep input & output ring-buffers  operating
correctly */

  } /* while */

 ALG_delete ( handle );
  fclose(infile); fclose(outfile);

  return 0;
}
```

6. Requirements for iDSP Components Compliance

It is the responsibility of the IDSPComponent developers to define and perform regression tests to ensure conformance with all the requirements listed below.

1. IDSPComponent fully conforms to ALL clauses listed in this section.

2. IDSPComponent fully conforms to all **mandatory requirements** of TI XDAIS standards. It must not use the optional feature of XDAIS called definition/implementation of the 'parent object'. It need **NOT** (but may choose to) conform to any other optional requirements of XDAIS. *It is a direct derivative of the ialg objects and handles defined in XDAIS.*

3. IDSPComponent provider plans and executes all tests needed to confirm IDSPComponent's full functionality. This testing is done in a PC with a PCI-based TI C6 DSP ("C6") board, and with C6's external memory located on this board. IDSPComponent's executable code is loaded in the C6 external memory using TI CCS-1.2 and JTAG interface

4. IDSPComponent design and/or implementation **do not** assume that, in a product environment, the IDSPComponent will run alone on a C6.

5. All code that runs on the DSP has to pass all regression tests (defined by the IDSPComponent developer) when compiled with at least the following compiler flags: –o3 –mtx –ml3

6. IDSPComponent is always in one of the two modes: Processing or Idle. The test environment alternates it between the two.

7. The test-environment of the IDSPComponent (i.e, the function 'main' as a proto-framework), defines and maintains input and output ring-buffers in the C6 external memory for data i/o -- (at least) one input and one output buffer for each data type.

8. (Test or other) environment will **not** switch the IDSPComponent from Idle to Processing state unless there is **at least** one frame worth of data in IDSPComponent's input ring buffer.

9. Input and Output ring-buffer(s)' pointers are managed by test/other environment. The IDSPComponent **intimates** to the test environment, the number of bytes consumed from the input ring-buffer (inputConsumed) and the number of bytes dumped to the output ring-buffer (outputProduced) at the end of each algorithm invocation.

10. In the production system, IDSPComponent does not input, output, process, or care about timing information. It just runs at a full speed, processing its input data to the output data. The IDSPComponent is not concerned with interface to a host or peripherals. All system data transfers for the algorithm to/from the host is managed by the iDSP platform. The IDSPcomponent however, must be capable of read/write wrapping, if required.

11. Once switched to Processing, IDSPComponent pulls exactly one full frame from its input ring buffer, processes it, and outputs the resulting frame into its output ring buffer – with other full or partial left-over frame(s) left untouched in the IDSPComponent's input ring

buffer. If an atomic unit of semantic input into the IDSPComponent is a vector of frames, each with its own input and/or output ring buffers and not a single frame (for example, a mux or demux vs. decoder), once switched, the IDSPComponent pulls one frame from each input buffer.

12. IDSPComponent is invoked is in the following generic manner (example for video decoder):

```
IDSPComponent.processbuffers ( IDSPComponent * handle,  IDSPRingBuffer*
                                       inbufs,
                              IDSPRingBuffer* outbufs,  IDSPResult*
                                       result)
{
   CompanyVideoDecoder ( VideoDecoderHandle handle,
                         IDSPRingBuffer*    input,
                         IDSPRingBuffer*    output,
                         IDSPResult*        result);
}
```

where: *input* → points to the starting address of valid data in the input ring-buffer.
output → points to the starting address of valid data in the output ring-buffer
handle → points to an object that contains all parameters required for the decoder.
result → points to the structure that contains the result of the invocation including all algorithm dependent status information (defined in a structure).

13. IDSPComponent provides all algorithm specific information that the component developer deems necessary to report back to the framework through the structure *IDSPResult*. If the necessary fields are not supported by the structure provided by the framework, the component should extend the *IDSPResult* structure to suit its needs.

14. All constant table data and contextual information required for the operation of the IDSPComponent is contained within the struct object pointed to by the *handle*. (Note: TI XDAIS standard mandates a specific mechanism to request internal memory to hold all these data. Even the memory required to hold the algorithm stack uses the same mechanism to get its share of internal memory. The mechanism is known as requesting memory through structure called 'memTab'(as discussed earlier and in the XDAIS standards).

15. (Test or other) environment invokes IDSPComponent by calling, for the example above, *CompanyVideoDecoder*.

16. IDSPComponent does not have any global and/or extern variables.

17. All variables in all functions within IDSPComponent are only either local or are an element in the IDSPComponent's XDAIS handle. Status and error conditions returned from the IDSPComponent are the element(s) of *IDSPResult* structure.

18. All IDSPComponent code is loaded into C6x's external memory. When running it, C6 uses the program cache.

19. The IDSPComponent must operate upon data in internal memory of C6x, and not on any data in its external memory. It is the responsibility of the IDSPComponent to page the relevant data into the internal memory before being operated on. This data is transferred between

DSP's internal and external memories using C6's DMA capabilities.

20. The IDSPComponent provider must confirm with TI, the product target DSP constraints on cache size available for IDSPComponent, and architect the IDSPComponent to conform to these limits. (Note: (a) Limit for IDSPComponent can be lower than overall cache size. (b) Product target DSP's cache size constraints can be different from cache size on the DSP you might be using for IDSPComponent development.)

21. IDSPComponent does not use more than a total of 32KB of C6 internal data memory, irrespective of the total size of the internal data memory on the C6 part.

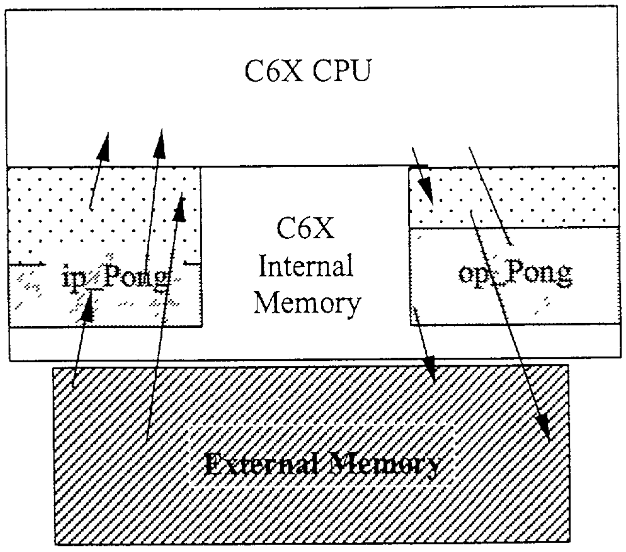
7. iDSP Component Performance Maximization

```
Requirements on Performance Maximization are not mandatory for  iDSP-Platform
compliance. However, these requirements are strongly recommended, as they
maximize the utilization of DSP's processing capacity - and thus the
competitive advantage of the "performance-maximized" iDSP-compliant
component, over another functionally equivalent and iDSP-compliant component
that is not "performance-maximized".
```

7.1. Data-Flow Architecture

DSP-performance-maximizing data-flow architecture within the IDSPComponent, between the input/output ring buffers in the external memory and the C6x internal memory is shown in Fig. 2.

- The decoder maintains a pair of ping-pong buffers, one pair each for input and output, within the internal C6x memory.
- Chunks of bit-stream data are DMA'ed into the pong-buffers while the C6x CPU operates on data in the ping-buffers and vice-versa. The same argument holds for the output side also.
- Since, the DSP performs the DMA operations in parallel with the CPU operations, the two operations may be executed in parallel. Theoretically, a performance gain of 2x can be achieved if the algorithm is structured in such a way that the time to transfer a semantic unit of data matches (at least, approximately) the time to process it.



— 'Ping' half of the double buffering .
Fig. 2
— 'Pong' half of the double buffering

Fig. 2

7.2. Instruction Cache Dependency

All executable code resides in the external memory. Instructions are cached into the Instruction-cache by the DSP during execution. The I-cache size varies depending on the specific C6 part. To reduce the probability of thrashing, the following actions are recommended:

- Functions that execute within a tight loop are grouped and linked into contiguous memory aligned on fetch-packet (32 byte) boundaries, which is optimal for TI C6 ISA family.
- In very rare cases, the call-stack depth exceeds the capacity of the Instruction-cache. In such cases, the source code might have to be logically partitioned so that the object code of these functions can be split logically so as to limit the depth of the call-stack to the amount of Instruction-cache.
- Consider using C6 compiler pragmas which let the developer chose the addresses for placing any object code. This is explained in detail in the **TMS3206000 Optimizing C-Compiler User's Guide (TI part No. spru187E).**

8. Integrating iDSP Components

8.1. Overview

Integration of IDSPComponents (algorithms) into the iDSP DSP runtime image is done through a single file. The iDSP Framework uses the file idspintrfc.c to create a task and the IO channels for each IDSPComponent instance. Each IDSPComponent is required to be XDAIS compliant, which means they are required to use the same interface functions. The main purpose of the idspintrfc.c file is to provide a central point to add IDSPComponents into the iDSP Framework and manage the instances of the IDSPComponents.

The iDSP Framework will create, invoke, and control each IDSPComponent through the XDAIS defined interface functions. Data delivery to/from the IDSPComponent is managed by the iDSP Framework. When the iDSP Framework invokes the IDSPComponent instance, its data will already be present. The IDSPComponent is restricted to processing/writing data passed to it through the data input/output pointers during the process_buffers call.

There are three different elements in the idspintrfc.c file that hold the information that the framework needs to invoke each IDSPComponent. The elements are the IDSPComponent Interface table, IDSPComponent IO Interface Channel table and the IDSPComponent Instance table. Currently all three tables are statically initialized at compile time, in later revision of the iDSP Framework a means to manage the Instance table dynamically during runtime will be provided.

8.2. The IDSPComponent Interface Table

An IDSPComponent's entry in the IDSPComponent Interface table first specifies its XDAIS compliant function table with its Vendor specific extensions. The table entry also holds other XDAIS requirements such as the IDSPComponent's parameters, command and the IDSPComponents IO requirements. The Interface table entry also holds pointers to a table of `IDSPChannelIntrfc` elements, this defines the IO channel specifics for the IDSPComponent. The interface table entries define the number of input/ouput channels and holds the pointers to the IO table for this IDSPComponent. Future revisions of the iDSP Framework will implement the params, sts and cmd of the IDSPComponent.

```
typedef struct IDSPComponentIntrfc {
  IDSPComponentFxns*   fxn_tbl;
  IALG_Params         *params;
  IALG_Status         *sts;
  IALG_Cmd            *cmd;
  unsigned int         num_input_chnls;
  IDSPChannelIntrfc   *input_chnls_req;
  unsigned int         num_output_chnls;
  IDSPChannelIntrfc   *output_chnls_req;
  char *               name;
} IDSPComponentIntrfc ;
```

`fxn_tbl` – Holds the pointer to an XDAIS compliant function table.

`param` – Holds the pointer for XDAIS compliant IDSPComponent parameter list. Currently not used in the iDSP Framework.

`sts` – Holds the pointer for the XDAIS compliant IDSPComponent's status structure. Currently not used in the iDSP Framework.

`cmd` – Holds the pointer for the XDAIS compliant IDSPComponent's command structure. Currently not used in the iDSP Framework.

`num_input_chnls` – Number of input channels needed by the IDSPComponent.

`input_chnls_req` – Pointer to the first IO Interface Channel Input object in the array of IO channel interfaces for an IDSPComponent.

`num_output_chnls` – Number of output channels needed by the IDSPComponent.

`output_chnls_req` – Pointer to the first IO Channel Output request object in the array of IO channel interfaces for an IDSPComponent.

`name` – Pointer to the full name of the component, such as

"TI MPEG-4 Video Decoder v 2.1",

An example below shows a filled out table which incorporates a G.723 decoder IDSPComponent and a G.723 Encoder.

```
IDSPComponentIntrfc IDSPComponentIntrfc_Table[] =
{
    &G723D_TI_FXNS,           //XDAIS compliant function tbl
     NULL,                     //
    NULL,                     //
    NULL,                     //
    1,                        //Num of Input Chnls
    g723_dec_ti_io_chnls,     //IO Chnl Interface table
    1,                        //Num of output chnls
    &g723_dec_ti_io_chnls[1],//IO Chnl interface table
/*-----------------------------------------------------*/
    &G723E_TI_FXNS,           //XDAIS compliant function tbl
    NULL,                     //
    NULL,                     //
    NULL,                     //
    1,                        //Num of Input Chnls
     g723_enc_ti_io_chnls,     //IO Chnl Interface table
    1,                        //Num of output chnls
    &g723_enc_ti_io_chnls[1],//IO Chnl interface table
/*-----------------------------------------------------*/
   NULL,                            // Un-init comp entry
   NULL,
   NULL,
    NULL,
   0,
   0,
   0,
   0
};
```

8.3. The IDSPComponent IO Interface Table

The IO requirements for a IDSPComponent are defined through an array of `IDSPChannelIntrfc` elements. There is a separate table for each IDSPComponent that defines the IO channels the IDSPComponent will need.

```
typedef struct IDSPChannelIntrfc {
  unsigned int num_of_bufs;  // fixed as 1 for now
  unsigned int buf_size;     //
  unsigned int chnl_dir;     //
  unsigned int swap_endian;  //
} IDSPChannelIntrfc;
```

Individual fields of the `IDSPChannelIntrfc` structure are described below.

`num_of_bufs` – This indicates the number of ring buffers for this channel. Currently the framework does not support this being greater than 1.

`buf_size` – This holds the number for the size of each ring buffer for this channel.

`chnl_dir` – Indicates if this particular channel is an input or an output channel. The two possible values for this field are:

`IDSPIntrfc_IO_INPUT__CHANNEL` – Input channel relative to DSP

`IDSPIntrfc_IO_OUTPUT_CHANNEL` – Output channel relative to DSP

`swap_endian` – Indicates if the data needs to have its endian swapped between the host and the DSP. The two values that are possible for this field are:

`IDSPIntrfc_IO_SAME_ENDIAN` – Endian is untouched during transfer

`IDSPIntrfc_IO_SWAP_ENDIAN` – Endian is swapped during transfer

In the example below is a Channel Interface IO table for a G.723 decoder and another table that defines the Channel Interface IO for a G.723 encoder. Note: In the `g723_dec_ti_io_chnls` table, the input channels are and must be grouped consecutively and at the start of the array. The output channels are then grouped consecutively following the input channels. Note also that the encoder table is constructed the same way.

```
IDSPChannelIntrfc g723_dec_ti_io_chnls[] =
{
  1,
  24,
  IDSPIntrfc_IO_INPUT__CHANNEL, // g723 dec Input channel
  IDSPIntrfc_IO_SAME_ENDIAN,
//-----------------------------------------------------------
  1,
  480,
  IDSPIntrfc_IO_OUTPUT_CHANNEL, // g723 dec Out channel
  IDSPIntrfc_IO_SAME_ENDIAN
};

IDSPChannelIntrfc g723_enc_ti_io_chnls[] =
{
  1,
  480,
  IDSPIntrfc_IO_INPUT__CHANNEL, // g723 enc In channel
  IDSPIntrfc_IO_SAME_ENDIAN,
//-----------------------------------------------------------
  1,
  24,
  IDSPIntrfc_IO_OUTPUT_CHANNEL, // g723 encoder Out channel
   IDSPIntrfc_IO_SAME_ENDIAN   };
```

8.4. The IDSPComponent Instance Table

The IDSPComponent Instance Table is used by the iDSP Framework to create an instance of an IDSPComponent that has been defined in the IDSPComponent Interface Table. An element in the IDSPComponent Instance Table holds a pointer to the entry of a IDSPComponent in the IDSPComponent Interface Table. The global variable `IDSPIntrfc_InstanceTable` must not be modified because this is the only means to tell the IDSP Framework which IDSPComponents to make instances of.

```
IDSPComponentIntrfc IDSPIntrfc_InstanceTable[] =
{
      &IDSPComponentIntrfc_Table[0],
      &IDSPComponentIntrfc_Table[1],
      NULL
};
```

The NULL element entry must be present so the iDSP Framework will terminate parsing of the instance table after the desired number of entries has been instantiated.

Currently IO Channel numbers for the DSP side of the IDSPComponents are generated in the order the IDSPComponents are entered in the Instance table starting with channel 1. Future revisions will add the capability to add/remove IDSPComponents dynamically to/from the Instance table. So in the example of this document, channels 1 and 2 would be assigned to the first element in the instance table, which is the G.723 IDSPComponent (a decoder IDSPComponent), and 3 and 4 would be assigned to the second element, which is the G.723 IDSPComponent (an encoder IDSPComponent).

8.5. Summary of Steps to Integrate A IDSPComponent

All the steps listed below are for elements that are defined in the idspintrfc.c file.

1. Insert all the necessary IDSPComponent information into the `IDSPComponentIntrfc_Table`.
2. Define a `IDSPChannelIntrfc` table to support the IO requirements for a IDSPComponent.
3. Add a pointer for each instance of the IDSPComponent to the `IDSPIntrfc_InstanceTable`.

9. Extending iDSP

The iDSP 2.0 developer can extend the iDSP framework by:

- Extending IDSPFormat, IDSPAudioFormat, or IDSPVideoFormat for new media formats
- Extending IDSPCommand for new command interfaces to IDSPComponents
- Extending IDSPPlugin for new types of plugins

The final version of this document will document implementation of these extensions for a baseline set of codecs in iDSP 2.0.

10. iDSP API Reference

This section defines and describes the types in the iDSP API.

10.1. IDSPParams

Constants

none

Typedef

```
typedef struct IDSPParams {
 int                number_inputs;
  int               number_outputs;
  IDSPFormat*       input_formats;
  IDSPFormat*       output_formats;

} IDSPParams;
```

Globals

none

Functions

none

A `IDSPParams` object holds the data needed to create an `IDSPPlugin` object. It can be created by the application, or obtained from `IDSPPluginManager_getRegistry()`. `IDSPParams` has the following fields:

- `number_inputs`
  The number of input tracks.
- `number_outputs`
  The number of output tracks.
- `input_formats`
  An array of IDSPFormat objects, one for each input. Each format object specifies the format that is carried on the input track.
- `output_formats`

  An array of IDSPFormat objects, one for each output. Each format object specifies the format that is carried on the output track.

10.2. IDSPFormat

Constants

```
#define     IDSPFormat_VIDEO            0
#define     IDSPFormat_AUDIO            1

#define     IDSPFormat_PARSED           0
#define     IDSPFormat_UNPARSED         1
```

Typedef

```
typedef struct IDSPFormat {
    int   format_type;
    char* format_name;
    int   data_type;

} IDSPFormat;
```

Globals

none

Functions

none

IDSPFormat is the base format type, with subtypes IDSPVideoFormat and IDSPAudioFormat. The fields of IDSPFormat are:

- format_type

  One of the constants IDSPFormat_VIDEO, IDSPFormat_AUDIO.

- format_name

  A name of the format, such as "H.263", "G.723", "MPEG-2".

- data_type

  One of the constants `IDSPFormat_PARSED`, `IDSPFormat_UNPARSED`. This indicates whether inputs to this plugin are frame-based chunks (see `IDSPPlugin_issue()/IDSPPlugin_reclaim()` below).

10.3. IDSPVideoFormat

Constants

```
#define     IDSPVideoFormat_COMPRESSED    0
#define     IDSPVideoFormat_PLANAR        1
#define     IDSPVideoFormat_PACKED        2
#define     IDSPVideoFormat_YUV111        4
#define     IDSPVideoFormat_YUV411        8
#define     IDSPVideoFormat_YUV420       16
#define     IDSPVideoFormat_YUV422       32
#define     IDSPVideoFormat_RGB          64
```

Typedef

```
typedef struct IDSPVideoFormat {
   IDSPFormat format;
   Int        video_type;
   int        frames_per_second;
   int        frame_width;
   int        frame_height;
   int        bits_per_pixel;
} IDSPVideoFormat;
```

Globals

none

Functions

none

IDSPVideoFormat provides the video specific parameters:

- `video_type`

  Either IDSPFormatVideo_COMPRESSED, or a boolean OR of IDSPVideoFormat_PLANAR or IDSPVideoFormat_PACKED and one of the YUV, RGB constants.

- `frames_per_second`

  Number of frames per second.

- `frame_width`

  Width of the video frame.

- `frame_height`

  Height of the video frame.

- `bits_per_pixel`

  Number of bits per pixel.

10.4. IDSPAudioFormat

Constants

```
#define   IDSPAudioFormat_COMPRESSED         0
#define   IDSPAudioFormat_PCM                1
#define   IDSPAudioFormat_SIGNED             2
#define   IDSPAudioFormat_UNSIGNED           4
```

Typedef

```
typedef struct IDSPAudioFormat {
   IDSPFormat format;
   Int        audio_type;
   int        microseconds_per_frame;
   int        sample_rate;
   int        number_channels;
   int        bits_per_sample;
   int        bits_per_second;
} IDSPAudioFormat;
```

Globals

none

Functions

none

IDSPAudioFormat provides the video specific parameters

- audio_type

  One of IDSPAudioFormat_COMPRESSED, and a Boolean OR of IDSPAudioFormat_PCM and either IDSPAudioFormat_SIGNED, IDSPAudioFormat_UNSIGNED.

- `microseconds_per_frame;`
- `sample_rate`
- `number_of_channels`
- `bits_per_sample`
- `bits_per_second`

10.5. IDSPPlugin

Constants

```
#define     IDSPPlugin_CODEC              0
#define     IDSPPlugin_RENDERER           1
#define     IDSPPlugin_MULTIPLEXOR        2
#define     IDSPPlugin_DEMULTIPLEXOR      3
#define     IDSPPlugin_SOURCE             4


#define     IDSPPlugin_INPUT              0
#define     IDSPPlugin_OUTPUT             1

#define     IDSPPlugin_OK                 0
#define     IDSPPlugin_TRACKNOTFILLED     1
#define     IDSPPlugin_ERROR              2
```

Typedef

```
typedef struct IDSPPlugin {
  char*               name;
  int                 type;
  IDSPParams          params;
  int                 processor;
} IDSPPlugin;
```

Globals

none

Functions

```
extern int IDSPPlugin getNumberTracks(IDSPPlugin* plugin,
                                      , int direction);
extern int IDSPPlugin_getDataSize(IDSPPlugin* plugin
                                        , int direction, int track);

  extern int IDSPPlugin_getTrackSize(IDSPPlugin* plugin
                                        , int direction, int track);
  extern IDSPFormat* IDSPPlugin_getFormat(IDSPPlugin* plugin,
                                          int direction, int track);
  extern int IDSPPlugin_connect(IDSPPlugin* plugin1
                                   , int output_track
                                   , IDSPPlugin* plugin2
                                   , int input_track);
  extern int IDSPPlugin_issue(IDSPPlugin* plugin, int direction
                                   , int track, unsigned char* data
                                   , int size);
  extern int IDSPPlugin_reclaim(IDSPPlugin* plugin,
                                   int direction, int track
                                   , unsigned char** data, int* size);

  extern int IDSPPlugin_control(IDSPPlugin* plugin,
                                   IDSPCommand* cmd,
                                   IALG_Status* status);
```

`IDSPPlugin` objects provide the runtime interface to `IDSPComponents`[2]. `IDSPPlugins` have input and output media tracks that carry blocks of data.

The fields of `IDSPPlugin are`

- `name`

  The complete name of the associated IDSPComponent, including make and version, such as "TI MPEG-4 Video Decoder v 2.5".

- `params`

  The attribute values associated with this plugin.

  - `plugin_type`

    A `plugin_type` is one of the constants `IDSPPlugin_CODEC`, `IDSPPlugin_RENDERER`, `IDSPPlugin_MULTIPLEXER`, `IDSPPlugin_DEMULTIPLEXER`, `IDSPPlugin_SOURCE`. Developers may extend the `framework` to include new plugin types.

  - `processor`

    The `processsor` is the processor ID that the IDSPPlugin's IDSPComponent is running on.

`IDSPPlugin_getNumberTracks()` returns the number of tracks for a given direction. Tracks are indexed from 0 for a plugin, so a plugin with two output tracks has tracks with ID 0 and 1.

`IDSPPlugin_getDataSize()` gets the largest data size that can be transferred on a track, where `direction` is one of `IDSPPlugin_INPUT`, `IDSPPlugin_OUTPUT`. `IDSPPlugin_getTrackSize()` returns the number of data buffers than can be issued before reclaimed. `IDSPPlugin_getFormat()returns the IDSPFormat object for a track.`

`IDSPPlugin_connect` connects the output track of one plugin to the input track of another.

IDSPPlugins use the *issue/reclaim* model on tracks. `IDSPPlugin_issue()` and `IDSPPlugin_reclaim()` are used as follows:

- `IDSP_issue`
  - `direction = IDSPPlugin_INPUT`

    A buffer with `size` amount of data is input for processing.
  - `direction = IDSPPlugin_OUTPUT`

    An empty buffer of size `size` is provided for output.
- `IDSP_reclaim`
  - `direction = IDSPPlugin_INPUT`

    An input buffer is released for the application to refill.
  - `direction = IDSPPlugin_OUTPUT`

    A buffer with processed data is returned. `*size` will contain the amount of data in the buffer.

When issue/reclaim is performed on *connected* tracks, the use is modified as follows:

- An issue is done on a connected output track, and a reclaim is done on the associated connected input track. The buffer pointers are ignored, and the return status provides information about the data transfer. This provides the ability for the application to manage dataflow through the IDSPPlugin graph.

One of following status variables is returned from issue:

- `IDSPPlugin_OK`

  Enough data has been transferred on media track to process.
- `IDSPPlugin_TRACKNOTFILLED`

  More data needed in media track.

A reclaim returns `IDSPPlugin_OK or IDSPPlugin_ERROR, if there was an error in receiving the data.`

`IDSPPlugin_control()` takes an IDSPCommand object and transfers it to the IDSPComponent. The actual object used in an application is a subclass of IDSPCommand defined by the component developer (see section ?). An extended IALG_Status [1] object is also passed to get the status result of the IDSPComponent's componentControl() call.

10.6. IDSPPluginManager

Constants

none

Typedef

```
typedef struct IDSPPluginManager {
};
```

Globals

none

Functions

```
extern  int IDSPPluginManager_initialize();
extern  int IDSPPluginManager_loadFramework(char* file
                                           , int processor);
extern  int IDSPPluginManager_startFramework(int processor);
extern  int IDSPPluginManager_stopFramework(int processor);
extern  int IDSPPluginManager_getRegistry(IDSPParams* params
                                           , int max_params);
extern  IDSPPlugin* IDSPPluginManager_createPlugin(
                                         IDSPParams* params);
extern  void        IDSPPluginManager_deletePlugin(
                                         IDSPPlugin* plugin);
extern  float IDSPPluginManager_getMemoryUtil(
                                         IDSPParams* params);
extern  float IDSPPluginManager_getProcessorUtil(
                                         IDSPParams* params);
```

The IDSPPluginManager provides the interface to general system functions of the iDSP Framework. In an executable or dynamic link module, IDSPPluginManager_initialize() is called before any other manager functions to initialize the iDSP Framework. IDSPPluginManager_loadFramework(), IDSPPluginManager_startFramework(), and IDSPPluginManager_stopFramework() are used to load, start and stop a particular framework image, which is in file, on a particular processor. IDSPPluginManager_getRegistry() returns the list of plugin attributes that can be used to create plugin objects. IDSPPluginManager_createPlugin() is used to create plugin objects, and IDSPPluginManager_deletePlugin() deletes them. IDSPPluginManager_getMemoryUtil() returns the memory utilization as a value between 0 and 1 of the plugin, and IDSPPluginManager_getProcessorUtil() returns the processor utilization. These values are used by application developers to manage DSP utilization resources.

10.7. IDSPRingBuffer

Typedef

```
typedef struct IDSPRingBuffer {
      unsigned char* LOW_MEM_PTR;
      unsigned char* HIGH_MEM_PTR;
      unsigned char* DATA_START;
      unsigned char* DATA_END;
} IDSPRingBuffer;
```

IDSPRingBuffer is the input/output interface between the iDSP-component (codec) and the iDSP-framework. Separate ring-buffers are defined both at the input and the output. The framework fills in the input ring-buffer and calls the component, which processes the input data and puts the output in another ring-buffer.

- LOW_MEM_PTR and HIGH_MEM_PTR are the lowest and highest memory addresses in the ring-buffer.
- DATA_START and DATA_END define the beginning and ending memory addresses of the data in the ring, which may wrap.

10.8. IDSPResult

Typedef

```
typedef struct IDSPResult {
      struct IDSPComponent* component;
      int*                  inputConsumed;
      int*                  outputProduced;
      int                   runStatus;
} IDSPResult;
```

The `IDSPComponent` updates this structure at the completion of its processing. It has the following fields:

`IDSPComponent*`

a pointer to the IDSPcomponent object instantiation.

`inputConsumed`

the number of 'bytes' consumed (from each input buffer) by the component after its most recent execution.

outputProduced

the number of 'bytes' output (to each output buffer) by the component after its most recent execution.

runStatus

one of the constants IDSP_ALG_COMPLETED, IDSP_ALG_UNDERFLOW

10.9. IDSPComponent

Typedef

```
typedef  struct   IDSPComponent {
      struct IDSPComponentFxns*   fxns;
} IDSPComponent;
```

IDSPComponent has one field, a pointer to its v-table of functions. IDSPComponent is an XDAIS algorithm.

10.10. IDSPComponentFxns

Typedef

```
typedef struct IDSPComponentFxns   {
      IALG_Fxns     ialg;
      Int (*processBuffers)(IDSPComponent*  alg,
                         IDSPRingBuffer*    inbufs[ ],
                           // input ring buffers
                         IDSPRingBuffer*    outbufs[ ],
                           // output ring buffers
                         IDSPResult*        result);
                           // result from processing
      Int (*componentControl)(IDSPComponent* alg,
                              IDSPCommand* cmd,
                        IALG_Status* status);
}   IDSPComponentFxns;
```

IDSPComponentFxns defines all the interface functions in the scope of the particular IDSPComponent in question. This is a direct extension of the IALG_Fxns defined by the XDAIS standard.

`processbuffers`	is the function call that starts the execution of the actual algorithm.
`IALG_Fxns`	-- a structure of function pointers defined in TI's XDAIS standard.
`IDSPRingBuffer`	-- defined in this section.
`IDSPResult`	-- defined in this section.

`componentControl` is the function used to send run-time commands (if any) to the IDSPcomponent (i.e, the algorithm).

`IDSPCommand*` -- a pointer to a programmer extended structure defined in this section.

`IALG_status*` -- a pointer to a programmer extended structure defined in XDAIS standard.

10.11. IDSPCommand

Typedef

```
typedef struct IDSPCommand   {

      Int size;

}
```

The type IDSPCommand as a base object. The programmer is free to extend this to include any run-time commands the component requires for its operation.

`size:` The size of this structure.

11. References.

1. TI expressDSP Algorithm Standard API Reference. Texas Instruments (SPRU360)
2. TI expressDSP Algorithm Standard Rules and Guidelines (SPRU352)